# 목차

# 구현 설명 – 개요 및 실행 방법

- 파이썬 3.8.5, 윈도우 환경에서 구현하였습니다.
- 내장 라이브러리 math만 사용하였습니다.
- 텍스트 파일의 이름+확장자명만 입력하면 자동으로 single, complete, average link에 대해서 계산합니다.

# 구현 설명 – 동작 방법

- 강의 PPT를 기반으로 만들었습니다.
- 동작 방식은 다음과 같습니다.
  - ① 각 좌표끼리 코사인 유사도를 계산한 2차원 리스트를 생성합니다.
  - ② 최대값의 index (i, j)를 알아내고, i열, j열에 대한 single/complete/average link를 계산해 새로운 열에 추가합니다. i행, j행에 대해서도 동일하게 처리해줍니다.
  - ③ 새로운 행렬을 추가해준 후 이전 i, j 행렬을 제거합니다.
  - ④ 최대값과 그 값의 index (i, j)를 각 Stack에 집어넣습니다.
  - ⑤ 행렬의 성분이 1개 남을 때까지 2번~4번 과정을 반복합니다.
  - ⑥ Stack의 top에 담긴 tree 형태의 cluster를 3개로 나눠줍니다.

# 구현 설명 – 동작 예시

- single link 예시입니다.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | x | 904 | 991 | 989 | 964 |
| B | 904 | x | 953 | 832 | 757 |
| C | 991 | 953 | x | 961 | 919 |
| D | 989 | 832 | 961 | x | 990 |
| E | 964 | 757 | 919 | 990 | x |

| | A | B | C | D | E | AC |
|---|---|---|---|---|---|---|
| A | x | 904 | 991 | 989 | 964 | x |
| B | 904 | x | 953 | 832 | 757 | 953 |
| C | 991 | 953 | x | 961 | 919 | x |
| D | 989 | 832 | 961 | x | 990 | 989 |
| E | 964 | 757 | 919 | 990 | x | 964 |
| AC | x | 953 | x | 989 | 964 | x |

| | B | D | E | AC |
|---|---|---|---|---|
| B | x | 832 | 757 | 953 |
| D | 832 | x | 990 | 989 |
| E | 757 | 990 | x | 964 |
| AC | 953 | 989 | 964 | x |

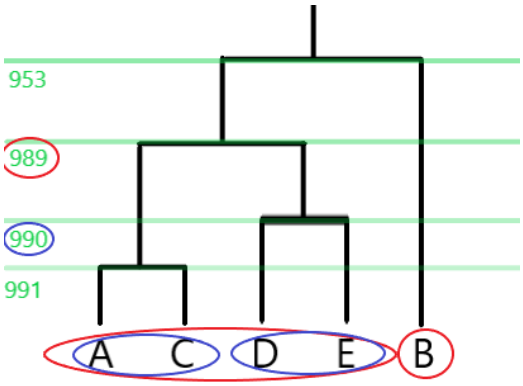
| | B | D | E | AC | DE |
|---|---|---|---|---|---|
| B | x | 832 | 757 | 953 | 832 |
| D | 832 | x | 990 | 989 | x |
| E | 757 | 990 | x | 964 | x |
| AC | 953 | 989 | 964 | x | 989 |
| DE | 832 | x | x | 989 | x |

| | B | AC | DE |
|---|---|---|---|
| B | x | 953 | 832 |
| AC | 953 | x | 989 |
| DE | 832 | 989 | x |

| | B | AC | DE | ACDE |
|---|---|---|---|---|
| B | x | 953 | 832 | 953 |
| AC | 953 | x | 989 | x |
| DE | 832 | 989 | x | x |
| ACDE | 953 | x | x | x |

| | B | ACDE |
|---|---|---|
| B | x | 953 |
| ACDE | 953 | x |

**[cluster tree]**




다음과 같은 span으로 3 cluster로 분리됩니다.

# 함수 설명 - Cosine Similarity

```python
def getCosSim(a, b):
    if(a==b):
        return -1

    dot = a[0] * b[0] + a[1] * b[1]
    norm = math.sqrt(a[0]**2+a[1]**2) * math.sqrt(b[0]**2+b[1]**2)

    return dot / norm
```

- 코사인 유사도를 반환하는 함수입니다.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

- 위 공식을 구현하였습니다.
- a, b가 다른 경우 범위는 (-1, 1)이 됩니다. 이 프로그램에서는 코사인 유사도가 높은 순으로 뽑아야 하기 때문에, 동일한 좌표가 들어온 경우 -1을 반환하도록 하였습니다.

# 함수 설명 – 사전 작업

## 변수

| 변수명 | 타입 | 설명 |
|---|---|---|
| fileName | string | 입력한 파일 이름 ex) Coord.txt |
| link | string | single, complete, average |
| dotList | list | tuple 형태로 좌표를 저장 |
| simList | list | 코사인 유사도를 저장, 초기 크기는 n by n |
| hierStack | list | tree의 node가 연결될 때 마다 생성, 각 단계별 tree를 중첩 list 꼴로 저장 |
| spanStack | list | node가 연결될 때의 코사인 유사도의 최댓값 저장 |
| rowMaxIdx | int | 가장 큰 유사도를 가진 행렬의 row index |
| colMaxIdx | int | 가장 큰 유사도를 가진 행렬의 col index |
| maxVal | double | 가장 큰 유사도 |
| newList | list | 새롭게 추가되는 리스트 ex) A,C -> AC |
| first | string | 3-cluster의 첫 번째 |
| second | string | 3-cluster의 두 번째 |
| third | string | 3-cluster의 세 번째 |

## 파일 읽기

```
f = open(fileName, 'r')
info = f.readline()
k, n = int(info.split()[0]), int(info.split()[1])
```

- 먼저 파일을 읽어, 첫 줄의 메타정보를 저장합니다.

```
lines = f.readlines()

# x,y 좌표값 리스트에 저장
for line in lines:
    x, y =  int(line.split(',')[0]), int(line.split(',')[1])
    dotList.append((x,y))
```

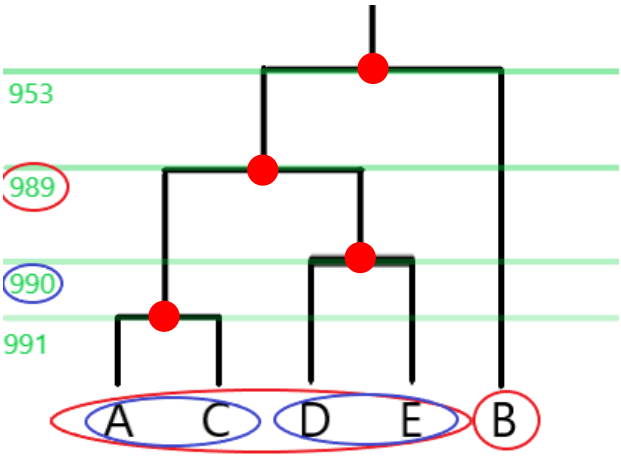
- 그 후 나머지 파일을 읽어 dotList에 tuple 형태로 점 좌표를 집어넣습니다.

```
simList = [[getCosSim(dotList[row], dotList[col]) for col in range(n)] for row in range(n)]
```

- cosine similarity는 n x n 형태로 저장합니다. 이 때 행렬의 (i, i) 성분은 getCosSim 함수에 의해서 -1로 저장됩니다.
- 행렬의 (i, j) 성분과 (j, i) 성분은 서로 대칭됩니다.

# 함수 설명 - Binary Tree 생성

- 이 작업은 다음과 같은 Tree가 있다고 할 때 각 과정(빨간 점)을 hierStack에 순서대로 저장합니다.




```
while(len(simList) > 1):

    if(len(simList)==2):
        hierStack.append([dotList[0],dotList[1]])
        spanStack.append(simList[0][1])
        break;
```

- 아래 과정들을 반복합니다.
- 리스트에 2개만 남은 경우 그 요소는 마지막 코사인 유사도와 0 뿐이기에, 각 위치를 지정하여 따로 저장하고 반복을 끝내도록 하였습니다.

```
rowMaxIdx = list(map(max,simList)).index(max(map(max,simList)))
colMaxIdx = simList[rowMaxIdx].index(max(map(max,simList)))

maxVal = simList[rowMaxIdx][colMaxIdx]
```

- 코사인 유사도의 최댓값과 그 index를 뽑았습니다.
- 각 유사도는 행렬의 대각선을 기준으로 대칭되어 있기에 필연적으로 rowMaxIdx가 colMaxIdx보다 작습니다.

*ex) (1,3), (3,1)에 찾는 값이 있다면 (1,3)을 찾게됨*

```
newList=[]
if(link == "single"):
    for row in range(len(simList)):
        simList[row].append(max(simList[row][rowMaxIdx], simList[row][colMaxIdx]))
        newList.append(max(simList[row][rowMaxIdx], simList[row][colMaxIdx]))
    newList.append(-1)
    simList.append(newList)

elif(link == "complete"):
    for row in range(len(simList)):
        simList[row].append(min(simList[row][rowMaxIdx], simList[row][colMaxIdx]))
        newList.append(min(simList[row][rowMaxIdx], simList[row][colMaxIdx]))
    newList.append(-1)
    simList.append(newList)

elif(link == "average"):
    for row in range(len(simList)):
        simList[row].append((simList[row][rowMaxIdx] + simList[row][colMaxIdx])/2)
        newList.append((simList[row][rowMaxIdx] + simList[row][colMaxIdx])/2)
    newList.append(-1)
    simList.append(newList)
```

- 두 노드를 연결해 새 노드를 형성합니다. 이 때의 코사인 유사도 값은 link의 방식에 따라서 다르게 계산합니다.
- single인 경우 두 노드 중 큰 값을 선택합니다.
- complete인 경우는 두 노드 중 작은 값을 선택합니다.
- average의 경우, 두 노드의 평균을 선택합니다.
- 행렬꼴로 코사인 유사도를 저장하기 때문에, 먼저 행 방향으로 하나씩 넣어주고, 열 방향으로 새 값들을 넣어줍니다.
- 값의 형태는 (n1, n2, n3, ... , nm, -1) 입니다. 마지막 원소는 자기 자신과의 값 즉 (i, i)번째 요소이므로 -1을 넣어줍니다.
- 

```
hierStack.append([dotList[rowMaxIdx],dotList[colMaxIdx]])
spanStack.append(maxVal)
```

- 저장된 Tree 꼴을 hierStack에 집어넣습니다.
- 마찬가지로 코사인 유사도의 max값 역시 spanStack에 집어넣습니다.

```
dotList.append([dotList[rowMaxIdx], dotList[colMaxIdx]])
del dotList[rowMaxIdx]
del dotList[colMaxIdx-1]
```

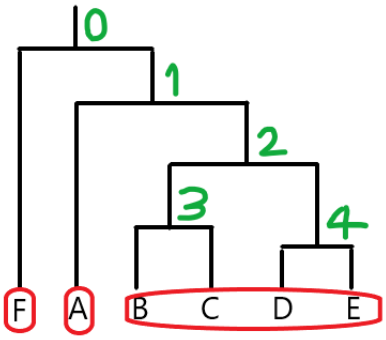
- dotList는 행렬표의 인덱스로 사용되기 때문에, 마찬가지로 list 꼴로 저장하고, 이전 값들은 삭제합니다.
- ex) A, C -> AC 라고 할 때, AC 넣은 후 A와 C 삭제
- 이 때 rowMaxIdx의 위치를 삭제하면, dotList의 index가 하나씩 앞당겨지게 되어 colMaxIdx에서 1을 뺀 값의 위치를 제거합니다.
- 이 과정까지가 반복되는 부분입니다.

```
hierStack.reverse()
```

- 모든 반복 과정이 끝난 후에는 편의를 위해서 hierStack의 순서를 바꿔줍니다.

# 함수 설명 – 3-Clustering

```
## Tree 순회, Clustering ##
first = hierStack[0][0]
second = hierStack[0][1][0]
third = hierStack[0][1][1]
```

- Binary Tree 형태로 만들어졌기 때문에 left child, right child, child's child의 3개로 clustering 할 수 있습니다.
- 이전의 tree를 만들 형태를 생각해보았을 때, hierStack[0]는 전체 tree의 모양이 담겨 있습니다.
- 

다음과 같은 형태라고 가정하면 hierStack[0]는 '[ F, [ A, [ [B,C], [D,E] ] ] ]'의 형태로 순차적으로 저장되어 있습니다.
- ```
hierStack.append([dotList[rowMaxIdx],dotList[colMaxIdx]])
spanStack.append(maxVal)
```
- 다음과 같은 코드로 hierStack이 형성되는데, 앞선 rowMaxIdx는 colMaxIdx보다 작습니다. 그러므로 점 2개가 묶이는 초기 상태가 아닌 경우에 colMaxIdx는 상대적으로 rowMaxIdx보다 나중에 형성된 것이고, dotList 혹은 hierStack의 형성 순서는 코사인 유사도가 작아지는 방향으로 되기 때문에, 결론적으로 트리가 우->좌 꼴로 커짐을 알 수 있습니다. 따라서 span 크기에 따라서 3-cluster로 구분한다면 hierStack[0]의 left child, right child's left child, right child's right child가 됩니다.
```

# 함수 설명 – 파일 쓰기

```
# Fir, Sec, Thi, String으로 변환 후 변경
first = str(first).replace('[','').replace(']','')
second = str(second).replace('[','').replace(']','')
third = str(third).replace('[','').replace(']','')
```

- text 파일에 쓰기 위해서 string 형태로 바꾸어 줍니다.
- 원본 Tree 구조를 담고 있기 때문에, 이를 없애 주기 위해 '[' , ']'를 제거해줍니다.

```
# 출력부
f2 = open(fileName[:-4]+"_output.txt", 'a')
f2.writelines('---\n')
f2.writelines(link+'\n')
f2.writelines('clusters: ['+first+'], ['+second+'], ['+third+']\n')
f2.writelines("span: "+ str(spanStack[-3])+ ', '+ str(spanStack[-2])+'\n')
f2.close()
```

- 정해진 양식에 맞게 파일에 출력해줍니다.
- 처음에 fileName을 확장자까지 받았으므로, index를 적절하게 조절합니다.
- spanStack은 hierStack과 동일하게 top+1, top+2 값에 해당하게 됩니다. (index로는 -2, -3)

# 함수 설명 – main

```
if __name__ == "__main__":
    print("Insert a file name in the same folder")
    fileName = input()
    hierarchy_clustering(fileName, "single")
    hierarchy_clustering(fileName, "complete")
    hierarchy_clustering(fileName, "average")
```

- main은 다음과 같습니다.
- link 종류에 맞게 3번 실행합니다.